# 3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

Рассмотрим подробно функционирование мобильного приложения. Для этого проведем анализ основных модулей программы и рассмотрим их зависимости. Также проанализируем все модули, которые входят в состав программного кода и рассмотрим назначение всех переменных, полей и методов этих модулей.

В состав мобильного приложения входят следующие модули:

* модуль взаимодействия с API;
* модуль представлений;
* модуль данных;
* модуль ресурсов

Большинство приложений под мобильную операционную систему Android содержат в себе один или несколько экранов, с которыми взаимодействует пользователь. В мобильном приложении, которое реализовывается в рамках данного дипломного проекта, также будет содержаться несколько таких экранов. Каждый экран приложения описывается классом, унаследованным от системного класса Activity.

У каждой Activity есть свой жизненный цикл, которым управляет операционная система. Это означает, что конкретный экран приложения может находиться в разное время в разных состояниях, в зависимости от того, как пользователь взаимодействует с этим приложением. Каждое такое состояние соответствует определенному этапу жизненного цикла. Класс Activity предоставляет возможность сообщать экрану приложения о том, что его состояние изменилось посредством методов обратного вызова, которые вызываются при соответствующих переходах на другой этап жизненного цикла Activity. Переопределяя эти методы в классах, которые описывают экран приложения, разработчик получает возможность контролировать переходы между различными этапами жизненного цикла, обрабатывая эти события необходимым ему образом.

Модуль представлений является модулем, содержащим в себе все Activity приложения, а также логику его поведения при переходах между различными этапами своего жизненного цикла.

## 3.1 Класс MainActivity. Основные компоненты.

При запуске мобильного приложения, первым делом пользователь попадает на главный экран приложения, представленный классом MainActivity. На этом экране у пользователя есть возможность воспользоваться следующим функционалом приложения:

* поиск медиаконтента;
* просмотр списка рекомендаций к просмотру;
* просмотр списка популярного на текущий момент медиаконтента;
* просмотр информации о личном профиле;
* переход к экрану аудита истории просмотров и оценок;
* переход к экрану изменение информации личного профиля;
* выход из личного профиля.

Класс MainActivity является наследником класса AppCompatActivity, который, в свою очередь, через другие иерархические связи, в конечном итоге наследуются от класса Activity. Благодаря такому наследованию, появляется возможность переопределить методы класса Activity, которые вызываются при переходах между этапами жизненного цикла главного экрана приложения.

Перед отображением экрана пользователю, вызывается метод onCreate(), в котором необходимо определить и проинициализировать элементы пользовательского интерфейса. Все элементы пользовательского интерфейса описаны в специальном модуле ресурсов в виде файлов в формате XML. Так, в модуле ресурсов находятся несколько пакетов, включая пакет layout, в котором хранятся ресурсы разметки пользовательского интерфейса для каждого экрана приложения.

В методе onCreate(), который переопределён в классе MainActivity, вызывается специальный метод супер-класса setContentView(), в который передается идентификатор ресурса, содержащего разметку необходимого экрана приложения. Для экрана MainActivity этот ресурс определяется идентификатором R.layout.activity\_main. Передав его в качества аргумента методу setContentView(), в качестве пользовательского интерфейса будет установлена разметка, содержащаяся в переданном ресурсе.

При запуске приложения на данном этапе, графический интерфейс успешно отобразится на экране мобильного устройства, однако для взаимодействия с компонентами этого интерфейса внутри программного кода, необходимо проинициализировать все его элементы в классе MainActivity. Для описания этих элементов используются специальные объекты, унаследованные от стандартного класса View. Эти элементы также инициализируются в методе onCreate(), который будет вызван при создании пользовательского экрана, до его отображения конечному пользователю. Для их инициализации необходимо объявить каждый компонент интерфейса, содержащийся в разметке R.layout.activity\_main, отдельным полем в глобальной области видимости класса.

Для MainActivity такими полями являются:

* private TabLayout tabLayout;
* private ViewPager viewPager;
* private ViewPagerAdapter adapter;
* private Toolbar toolbar;
* public SearchView searchView.

Поля класса ассоциируются с компонентами интерфейса из разметки с помощью метода Activity findViewById(). В качестве аргумента этот метод принимает уникальный идентификатор компонента, определенный в модуле ресурсов. Таким образом связываются все компоненты пользовательского интерфейса во всех классах приложения, описывающих Activity.

Одним из компонентов MainActivity является элемент SearchView. Этот элемент используется для организации удобного поиска медиаконтента в приложении. SearchView инициализируется в отдельном этапе жизненного цикла приложения, который вызывает метод обратного вызова onCreateOptionsMenu(Menu menu). Обусловлено это тем, что этот элемент является специальным компонентом, называемым меню. Такие меню описываются специальным одноименным пакетом в модуле ресурсов, также в виде разметки XML.

Этапы инициализации компонента SearchView:

* вызов специального метода Activity getMenuInflater(), который возвращает объект типа MenuInflater, использующийся для инициализации ресурса меню;
* вызов метода inflate() объекта MenuInflater, на вход которому подается файл ресурса, в котором описывается меню;
* инициализация объекта MenuItem, описывающего меню;
* инициализация SearchView путем приведения типа объекта MenuItem к типу SearchView.
* установка обработчика события на компонент.

В методе onCreate() также инициализируются и другие компоненты, использующиеся в реализации всей необходимой функциональности главного экрана, в том числе и фрагменты, о которых будет подробно сказано в разделе 3.2.

## 3.2 Класс MainActivity. Фрагменты главного экрана приложения.

В MainActivity реализована большая часть пользовательского функционала. Все элементы для его использования очень трудно разместить на одном экране, так как на это просто не хватит пространства экрана смартфона. Для решения этой проблемы было решено использовать так называемые Фрагменты (Fragments). Фрагменты, как и Activity, представляют собой контейнеры, в которых можно разместить элементы пользовательского интерфейса. Однако использование фрагментов позволяет не отображать их на экране до тех пор, пока это не потребуется пользователю. В нужный момент, который определяет разработчик специальной логикой, фрагменты вставляются в определенные места Activity, отображаясь на экране для пользователя. Места, в которые вставляются фрагменты – это специальные контейнеры, одним из которых является ViewPager, который и используется в MainActivity.

Для комфортной навигации между фрагментами через элемент ViewPager, также был добавлен контейнер TabLayout, объявленный в классе MainActivity. Этот контейнер представляет собой удобное для пользователя меню навигации между фрагментами, отображая на специальной панели имя текущего отображаемого фрагмента (рисунок 3.1).

ViewPager отвечает за показ фрагментов и их прокрутку, используя специальный адаптер ViewPagerAdapter. Этот адаптер предоставляет элементу ViewPager информацию о том, какой именно фрагмент в текущий момент времени должен быть отображен в нем, анализируя поведение пользователя (пользователь взаимодействует с этой областью экрана, проводя пальцем или стилусом вправо, или влево по нему).

Фрагменты, которые используются на экране MainActivity приложения:

* MovieSearchFragment;
* UserProfileFragment;
* RecommendationsFragment.

Фрагменты, как и Activity, должны наследоваться от специального класса. Для них таким классом является класс Fragment. При обнаружении класса, который наследуется от Fragment, операционная система понимает, какое именно поведение задать этому компоненту. Фрагменты очень похожи на Activity:

* инициализация проводится также, как и для Activity;
* графические интерфейс также описывается в модуле ресурсов;
* имеют свой личный жизненный цикл.

После инициализации всех фрагментов, можно проводить инициализацию специального адаптера ViewPagerAdapter и отображать фрагменты на главном экране приложения с помощью элемента ViewPager. Этапы инициализации фрагментов и отображения их в ViewPager:

* объявление и инициализация объектов типа Fragment;
* объявление и инициализация объекта ViewPager, с помощью вызова метода Activity findViewById(), в который передаётся уникальный идентификатор компонента ViewPager;
* объявление и инициализация объекта ViewPagerAdapter при помощи передачи в его конструктор объекта FragmentManager, который можно получить при помощи вызова метода Activity getSupportFragmentManager();
* добавление фрагментов в адаптер ViewPagerAdapter при помощи последовательных вызовов его метода addFragment(), в качестве параметров которому передаются два аргумента: один из фрагментов, необходимый к отображению в ViewPager, а также строка, содержащая заголовок (название) этого фрагмента, который будет отображаться пользователю;
* установка адаптера в ViewPager при помощи вызова его метода setAdapter(), в который передается ViewPagerAdapter.
* инициализация TabLayout при помощи вызова его метода setupWithViewPager(), в который передается инициализированный элемент ViewPager;
* вызов метода setSupportActionBar(), в который передается объект ToolBar пользовательского интерфейса, являющегося «шапкой» приложения, и содержащего заголовок (название) приложения, а также элементы меню.

После инициализации MainActivity и его фрагментов, главный экран приложения готов к отображению конечному пользователю (рисунок 3.2.1).



Рисунок 3.1 – Главный экран приложения после инициализации компонентов.

3.3 Фрагмент MovieSearchFragment.

Как было сказано ранее, фрагменты обладают жизненным циклом также, как и Activity. Инициализацию фрагмента и связку его компонентов графического интерфейса с его файлом ресурсов, нужно проводить на самом раннем этапе жизненного цикла фрагмента, который определяется вызовом метода onCreateView().

Полями класса MovieSearchFragment являются:

* protected static RecyclerView recyclerView;
* SwipeRefreshLayout swipeRefreshLayout;
* public static List<Movie> moviesList;
* protected static List<Movie> filteredMovieList;
* static MovieApi apiService;
* static MovieRecyclerViewAdapter recyclerViewAdapter;
* static boolean loading;
* ProgressBar progressBar;

Объект RecyclerView является компонентом, который отвечает за отображения списка фильмов в фрагменте. Этот элемент обладает крайне гибкой настройкой, о которой будет сказано ниже. Каждый элемент такого списка – это отдельный компонент View, файлом ресурса которого является файл res/layout/item\_movie.xml. Внешний вид проинициализированного элемента представлен на рисунке 3.2.

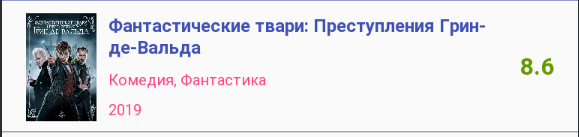


Рисунок 3.2 – Элемент списка res/layout/item\_movie.xml.

Таких элементов в списке может быть много, и за загрузку элементов в список RecyclerView отвечает специальный адаптер, который представлен отдельным классом MovieRecyclerViewAdapter, являющийся наследником класса RecyclerView.Adapter<>.

Класс RecyclerView.Adapter<>, как и все классы, для которых он является базовым, предоставляет возможность динамически заполнять контейнер RecyclerView элементами списка (рисунок 3.3.1), инициализируя их элементы графического интерфейса определенным образом, используя специальный класс ViewHolder.

MovieRecyclerViewAdapter располагается в пакете приложения bsuir.study.movier.adapter и работает в паре с MovieHolder, который располагается в пакете bsuir.study.movier.adapter.holder. MovieHolder является наследником класса ViewHolder.

Одним из полей класса MovieRecyclerViewAdapter является коллекция List<Movie> moviesList, которая хранит данные о медиаконтенте, которыми будут заполняться элементы списка в RecyclerView.

Наследуясь от RecyclerView.Adapter<>, класс MovieRecyclerViewAdapter должен переопределить все методы своего базового класса. Есть несколько методов, обязательных для переопределения:

* Public MovieHolder onCreateViewHolder(ViewGroup parent, int viewType);
* public int getItemViewType(int position);
* public void onBindViewHolder(MovieHolder movieHolder, final int position);
* public int getItemCount().

Метод onCreateViewHolder() отвечает за инициализацию компонента View, являющегося элементом списка, и определяет его разметку из модуля ресурсов (рисунок 3.3.1). Этот метод возвращает новый объект MovieHolder, содержащий внутри себя инициализированные компоненты графического интерфейса, готовые к дальнейшему заполнению.

Метод getItemViewType() возвращает уникальный идентификатор ресурса R.layout.item\_movie, указывающего на файл с разметкой одиночного элемента списка.

Метод getItemCount() возвращает кол-во элементов, которые в конечном итоге должны быть загружены в RecyclerView. Это количество определяется количеством элементов в коллекции данных List<Movie> moviesList.

Самым важным методом в классе MovieRecyclerViewAdapter, является метод onBindViewHolder(MovieHolder movieHolder, final int position). Именно этот метод отвечает за заполнение элемента списка данными. Он принимает два параметра в качестве аргументов вызова:

* объект MovieHolder, инициализированный методом onCreateViewHolder();
* позиция элемента в коллекции данных List<Movie>, по которому следует обращаться к коллекции для получения информации о медиаконтенте.

В теле этого метода происходит заполнение компонентов пользовательского интерфейса элемента списка MovieHolder данными, такими как заголовок, жанр, год, а также постер. Постер фильма в элементе списка представляет собой View-компонент ImageView, в который помещается изображение.

Хранить все изображения на устройстве не представляется возможным, ввиду его огромного количества и постоянного обновления. Эта проблема решается с помощью специальной библиотеки Picasso, которая позволяет загрузить изображение из сети Интернет по его URL. Такое заполнение происходит посредством последовательных вызовов статических методов библиотеки:

* with(), в который передается текущий контекст приложения;
* load(), в который передается URL изображения для загрузки;
* placeholder(), в который передается идентификатор из модуля ресурсов, указывающий на изображение, которое будет отображаться по умолчанию в элементе ImageView до того, как произойдет полная загрузка изображения из сети Интернет;
* into(), в который передается идентификатор элемента ImageView из модуля ресурсов, в который необходимо произвести загрузку изображения.

Также в этом методе происходит регистрация обработчика события касания для каждого элемента списка, посредством вызова метода movieHolder.itemView.setOnClickListener(onClickListener), принимающий в качестве единственного аргумента объект типа View.OnClickListener, который необходимо предварительно инициализировать. Инициализация такого объекта происходит благодаря использованию анонимного класса. Анонимный (безымянный) класс объявляется без задания имени класса и переменных данного безымянного типа – задаётся только конструктор класса вместе с его реализацией. У анонимного класса может быть только один экземпляр, причём он создаётся сразу при объявлении класса. Поэтому перед объявлением анонимного класса следует ставить оператор new.

Переопределяя метод этого класса onClick(), можно определить поведение системы при возникновении события нажатия на элемент списка. Дальнейшим поведением системы после такого события является запуск экрана приложения, содержащего детальную информацию о медиаконтенте, по элементу списка которого это событие произошло (рисунок 3.3).

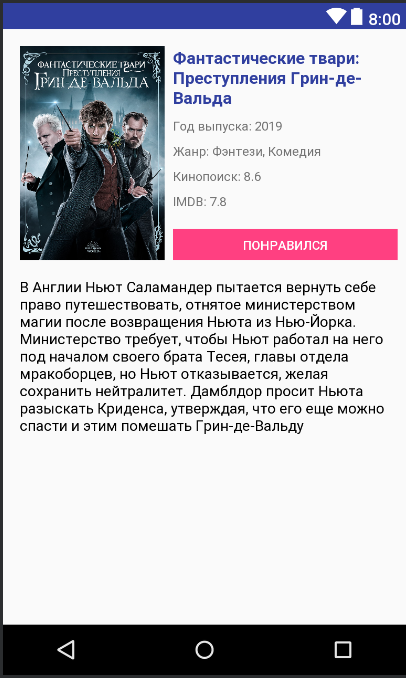


Рисунок 3.3 – Экран приложения для просмотра детальной информации о медиаконтенте.

## 3.4 Данные для MovieSearchFragment. Класс Movie.

Одним из полей класса MovieSeacrhFragment является коллекция List<Movie> moviesList. Данная коллекция, как было описано выше, является хранилищем данных о медиаконтенте, который необходимо отобразить пользователю в виде списка RecyclerView. Эта коллекция содержит в себе произвольное количество объектов Movie, которые представляют модуль данных в структурной схеме приложения,

Класс Movie расположен в пакете bsuir.study.movier.model и содержит следующие поля, помогающие максимально широко определить модель данных:

* private int id – содержит уникальный идентификатор медиаконтента в базе постащика контента;
* private String title – содержит название медиаконтента (фильма, сериала и др.)
* private String rating – содержит рейтинг медиаконтента;
* private String overview – содержит краткое описание для объекта медиаконтента;
* private String genre – содержит жанр, который представляет объект медиаконтента;
* private String kinopoiskRating – содержит рейтинг по версии ресурса «Кинопоиск»;
* private String imdbRating – содержит рейтинг по версии ресурса “iMDB”;
* private String posterImgUrl – содержит URL изображения постера для медиаконтента;
* private int year – содержит год выпуска объекта медиаконтента.

Все вышеперечисленные поля можно инициализировать с помощью конструктора класса, который принимает значения для этих полей в качестве своих аргументов. Все поля класса объявлены с использованием модификатора доступа private, что означает что доступ к этим полям для изменения возможет только изнутри этого объекта. Ни один другой компонент приложения не имеет доступ к модификации данных, содержащихся в этих объектах. Для их изменения используются специальные методы с префиксами «get» и «set», которые определены внутри класса Movie. Так как методы определены внутри класса, они имеют доступ для изменения значений полей класса.

Для получения информации о медиаконтенте, используется специальный модуль взаимодействия с API, который вызывается из других модулей приложения по необходимости. Данные о медиаконтенте приходят в мобильное приложение в формате JSON (рисунок 3.4). JSON основан на двух структурах данных:

* коллекция пар ключ/значение. В разных языках, эта концепция реализована как объект, запись, структура, словарь, хэш, именованный список или ассоциативный массив;
* упорядоченный список значений. В большинстве языков это реализовано как массив, вектор, список или последовательность.





Рисунок 3.4 – Схематичное изображение формата JSON.

Для обработки данных, которые пришли в модуль взаимодействия с API от поставщика контента, используется библиотека Gson, которая помогает обрабатывать строку в формате JSON, конвертируя ее в необходимые Java-объекты и наоборот. Такая конвертация называется «сериализация» и «десериализация» соответственно.

Как при сериализации, так и при десериализации, Gson использует ряд аннотаций, которыми помечаются поля класса Java-объекта. В классе Movie поля помечены аннотациями @SerializedName и @Expose.

Аннотация @ SerializedName позволяет указать библиотеке Gson имя ключа, которое она будет искать в JSON-строке при десериализации объекта. Если не указывать эту аннотацию, то библиотека по умолчанию будет искать ключ, соответствующий названию переменной поля класса, в который эта строка будет десериализовываться.

Аннотация @Expose работает только в паре с методом GsonBuilder.excludeFieldsWithoutExposeAnnotation(), который исключает из обработки все поля, не имеющие аннотации @Expose. Если не использовать такой подход, то по умолчанию Gson будет производить попытки десериализовать JSON-строку для всех полей класса, что не всегда удобно, так как некоторые поля класса могут быть служебными, ожидать которых в пришедшей JSON-строке не приходится.

Указав и сконфигурировав все необходимые аннотации для полей класса, можно производить запросы через модуль взаимодействия с API к поставщику контента и, используя библиотеку Gson, создавать объекты Movie, помещая их в последствии в коллекцию List<Movie> moviesList, которая используется в качестве исходных данных для создания и отображения списка медиаконтента конечному пользователю.

3.5 Взаимодействие с API. Класс APIClient и интерфейс MovieApi.

API (программный интерфейс приложения, интерфейс прикладного программирования) (англ. application programming interface) - описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой.

Модуль взаимодействия с API отвечает за отправку HTTP запросов к серверу поставщика контента, на котором находится стороннее API, предоставляющее разностороннюю информацию и медиаконтенте.

Все запросы формируются и отправляются в соответствии с протоколом HTTP. При таком способе общения с сервером, все запросы отправляются по определенному URL, каждый из которых соответствует определенному методу API.

Класс APIClient отвечает за отправку HTTP запросов к поставщику контента. Для этого используется библиотека Retrofit, позволяющая организовать работу с API, не прибегая при этом к системным вызовам функций Android. Такой подход очень удобен, так как при этом разработчику не приходится заботиться о правильной структуре HTTP запроса, а лишь описать интерфейс общения с API. Такой интерфейс в дипломном проекте описывается java-интерфейсом MovieApi.

Класс APIClient содержит в себе два поля:

* private static final String BASE\_API\_URL = "https://api.themoviedb.org/";
* private static Retrofit retrofit = null.

У всех URL которые используются для запросов к API, можно выделить общую часть, с которой начинаются все URL, в независимости от метода, к которому происходит обращение. Для выбранного API такую общую часть также можно выделить. Эта общая часть URL указана в поле класса BASE\_API\_URL. Это поле объявлено как статическое и константное, что означает невозможность его изменения программными вызовами. Это гарантирует его однозначность во всех модулях системы.

Вторым полем класса APIClient является переменная типа Retrofit. Это класс одноименной библиотеки, описанной выше. Эта переменная инициализируется в методе public static Retrofit getClient(), вызов которой возвращает проинициализированное значение переменной этого типа, подготавливая тем самым библиотеку Retrofit к работе.

Для корректной работы с Retrofit понадобятся три класса:

* POJO (Plain Old Java Object) – этим классом является класс Movie;
* Retrofit - класс для обработки результатов. Этим классом является описанный выше класс APIClient. Ему нужно указать базовый адрес в методе baseUrl(), которым является поле BASE\_API\_URL класса APIClient;
* Interface - интерфейс для управления адресом, используя команды GET и POST. Этим интерфейсом в приложении является интерфейс MovieAPI.

В интерфейсе MovieAPI необходимо описать методы для взаимодействия Retrofit с поставщиком контента по определенным правилам. Так, используются следующие аннотации, которые характеризуют метод запроса к серверу API поставщика контента (таблица 3.1).

Таблица 3.1 – Описание аннотаций Retrofit.

|  |  |
| --- | --- |
| Аннотация | Описание |
| @GET() | GET-запрос для базового адреса. Также можно указать параметры в скобках. |
| @POST() | POST-запрос для базового адреса. Также можно указать параметры в скобках. |
| @Path | Переменная для замещения конечной точки, например, username подставится в {username} в адресе конечной точки. |
| @Query | Задаёт имя ключа запроса со значением параметра. |
| @Body | Используется в POST-вызовах (из Java-объекта в JSON-строку). |

Для аннотации @GET в скобках указывается дополнение к базовому адресу URL. Аналогичное дополнение указывается и для аннотации @POST().

Аннотация @Query используется перед атрибутами методов MovieApi, если необходимо подставить значение атрибута в URL, составленный аннотацией @GET(). Используя эти правила, был описан метод MovieApi, позволяющий обратиться к поставщику контента за информацией о контенте, фильтруя его специальным параметром «query», в который указывается искомое название контента или его описание. Этот метод используется для асинхронного обращения к API, не блокируя при этом главный поток приложения, обрабатывающий и отрисовывающий пользовательский интерфейс.